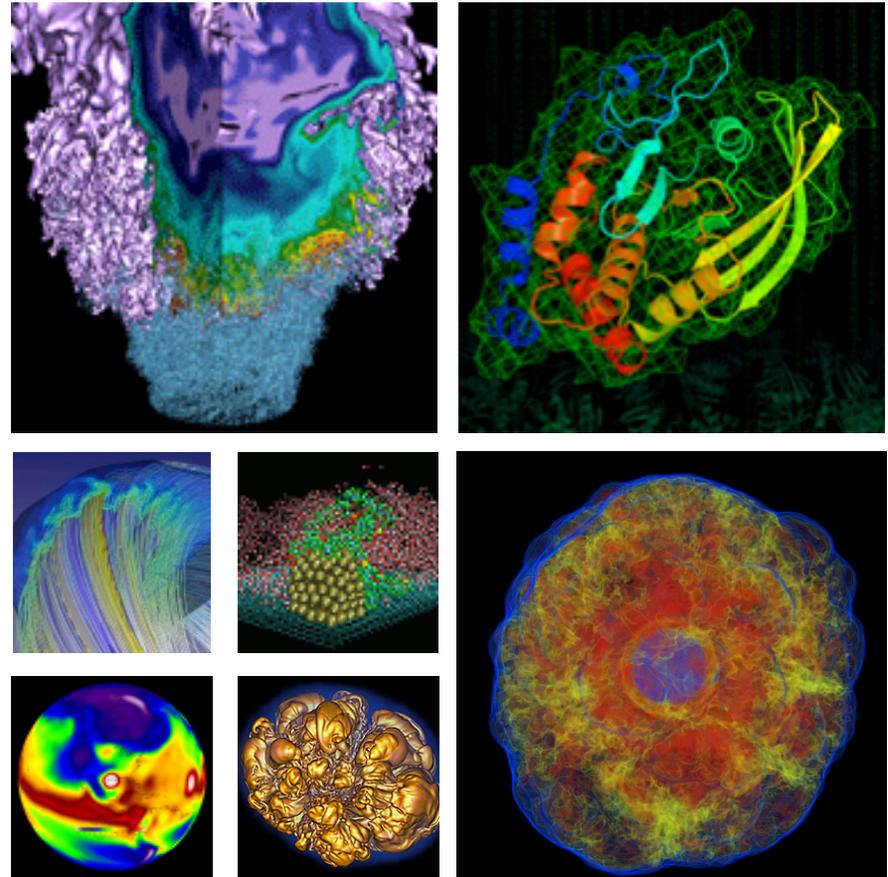


Parallel Debugging Tools

New User Training 2017



Woo-Sun Yang
User Engagement Group, NERSC

February 23, 2017

Debugging



- **Why debugging?**
 - Your program crashes for a unknown reason
 - Your program gives wrong results
- **How to find coding errors?**
 - Using print statements
 - Insert print statements in strategic locations
 - Can be difficult to know where the code fails and whether variables have incorrect values
 - Recompile whenever you make a change - tedious and time-consuming
 - Using debuggers
 - You compile only once (generally)
 - Can point to where the code fails
 - They let you control execution pace of your program and examine variables
 - Useful tools can aid your detective work greatly
 - Visualization and statistics
 - Memory debugging
 - MPI message queue

Parallel debuggers on Cori and Edison



- **Parallel debuggers with a graphical user interface**
 - DDT (Distributed Debugging Tool)
 - TotalView
- **Specialized debuggers on Cori and Edison**
 - STAT (Stack Trace Analysis Tool)
 - Collect stack backtraces from all (MPI) tasks
 - ATP (Abnormal Termination Processing)
 - Collect stack backtraces from all (MPI) tasks when an application fails
- **Valgrind**
 - Suite of debugging and profiling tools

DDT and TotalView



- **GUI-based traditional parallel debuggers**
 - Intuitive and simple to use; many useful tools
 - Allow to control program's execution pace and, sometimes, execution path
 - Set breakpoints, watchpoints and tracepoints
 - Display the values of variables and expressions, and visualize arrays
 - Check whether the program is executing as expected
 - Memory debugging
 - Message queue feature **NOT** working with Cray MPI
- **Works for C, C++, Fortran programs with MPI, OpenMP, pthreads**
 - DDT supports CAF (Coarray Fortran) and UPC (Unified Parallel C), too
- **Maximum application size for the debuggers at NERSC**
 - DDT: up to 4096 MPI tasks on Cori (Haswell and KNL) and Edison
 - TotalView: up to 512 MPI tasks on Cori (Haswell) and Edison
 - Licenses shared among users and machines
- **For info**
 - <https://www.allinea.com/products/ddt>
 - <http://www.nersc.gov/users/software/debugging-and-profiling/ddt/>
 - <http://www.roguewave.com/products/totalview>
 - <http://www.nersc.gov/users/software/debugging-and-profiling/totalview/>

How to build and run with DDT

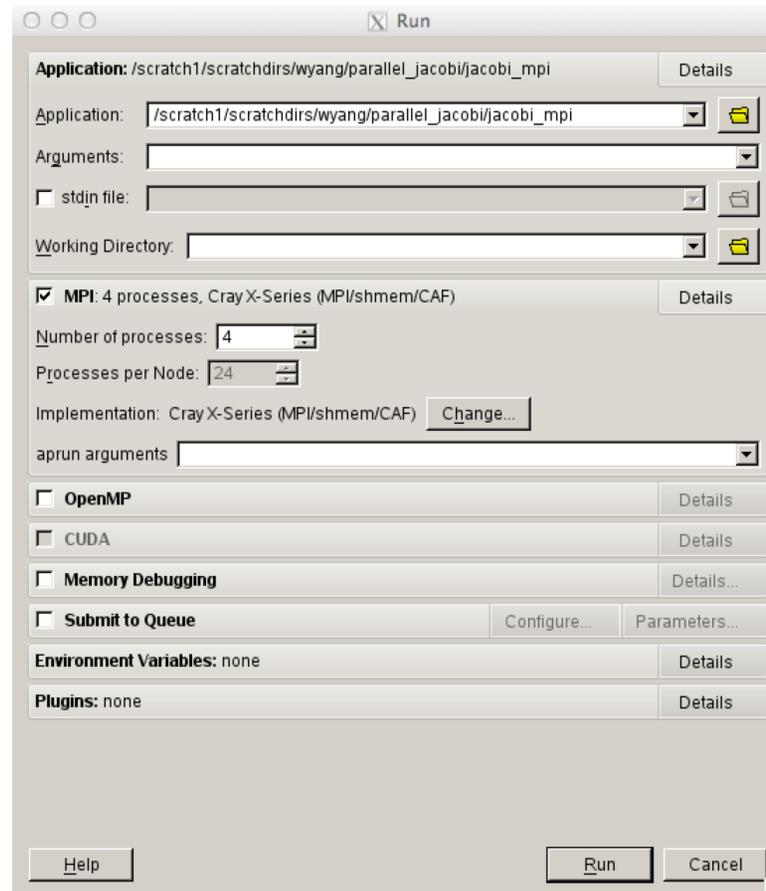


\$ **ftn -g -O0 -o jacobi_mpi jacobi_mpi.f90** Compile with -g to have debugging symbols
Include -O0 for the Intel compiler

\$ **salloc -N 1 -t 30:00 -p debug -C knl,quad,cache** Start an interactive batch session

\$ **module load allineatools** Load the allineatools module to use DDT

\$ **ddt ./jacobi_mpi** Start DDT



The module name will change to 'forge' for future versions

If you are far away from NERSC



- Remote X window application (GUI) over network: slow response
- Two solutions
 - Use NX to improve the speed
 - Works with any X window applications
 - <https://www.nersc.gov/users/network-connections/using-nx/> (general)
 - http://portal.nersc.gov/project/mpccc/nx/NX_Tutorial/Start_Over.html (installation and quick user guide)
 - Use Alinea Forge remote client
 - Runs on your desktop/laptop
 - Submit a debugging batch job from a NERSC machine and make the client **reverse connect** to the job
 - Displays results in real time
 - No license file required on your local desktop/laptop
 - <http://www.allinea.com/products/forge/download> (downloading remote clients)

Using NX



The screenshot displays the Allinea DDT - Allinea Forge 7.0 IDE interface. The main window shows the source code for 'jacobi_mpi.f90' with the following visible lines:

```
39 ! Allocate memory for arrays.
40
41 allocate(u(0:n,js-1:je+1), unew(0:n,
42
43 ! Initialize f, u(0,*), u(n:*), u(*,0)
44
45 call init_fields(u,f,n,js,je)
46
47 ! Main solver loop.
48
49 h = 1.0 / n
50
51 do k=1,maxiter
52 call mpi_sendrecv(u(1,js ),n-1,mp
```

The 'Locals' window on the right shows the following variables and values:

Variable Name	Value
f	---
je	1499
js	0
n	23999
u	---

The 'Stacks' window at the bottom shows the current process:

Processes	Function
16	jacobi_mpi (jacobi_mpi.f90:45)

Using Allinea remote client



(1) Select 'Configure' to create a configuration for a NERSC machine

2nd entry for a MOM node
Cori: cmom02 or cmom06
Edison: edimom01, ..., or edimom06

(2) Create a configuration

Connection Name: cori

Host Name: wyang@cori.nersc.gov wyang@cmom02.nersc.gov

Remote Installation Directory: /usr/common/software/allineatools/default

Remote Script: /usr/common/software/allineatools/remote-init

Always look for source files locally

Test Remote Launch

Help OK Cancel

Note that the paths will change for future versions

Using Alinea remote client (Cont'd)



(3) Select a machine

RUN

Run and debug a program.

ATTACH

Attach to an already running program.

OPEN CORE

Open a core file from a previous run.

MANUAL LAUNCH (ADVANCED)

Manually launch the backend yourself.

OPTIONS

Remote Launch:

✓ Off

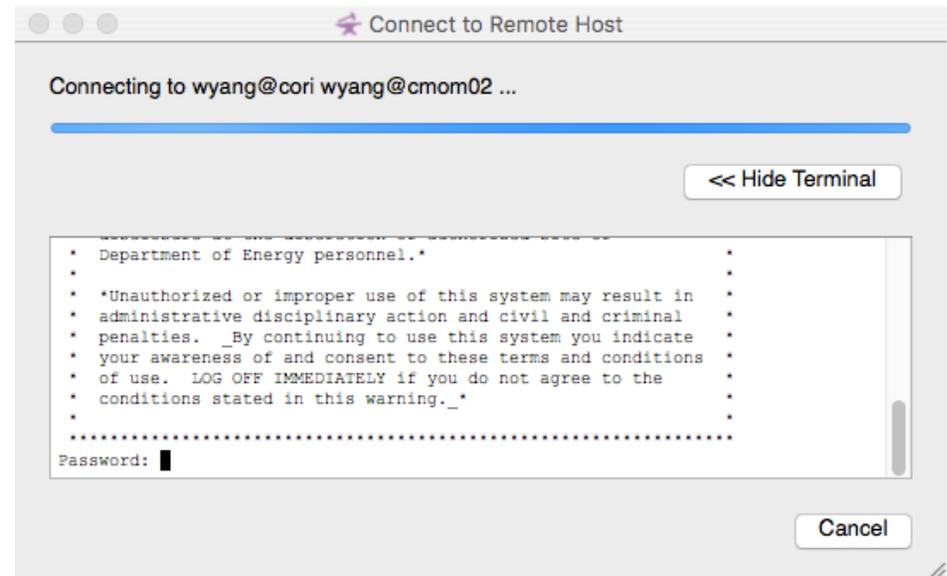
Configure...

cori

edison

carl

(4) Enter the NIM password



Using Alinea remote client (Cont'd)



(5) Submit a batch job on a NERSC machine and start DDT

```
$ salloc -N 1 -t 30:00 -p debug -C knl
...
$ module load allineatools
$ ddt --connect ./jacobi_mpiomp
```

(6) Accept the request



A new Reverse Connect request is available from nid08791 for Alinea DDT.

Command Line: --connect ./jacobi_mpiomp

Do you want to accept this request?

Help Accept Reject

(7) Set parameters and run

Application: /global/cscratch1/sd/wyang/debugging/jacobi_mpiomp Details

Application: /global/cscratch1/sd/wyang/debugging/jacobi_mpiomp ▼ 📁

Arguments: ▼

stdin file: ▼ 📁

Working Directory: ▼ 📁

MPI: 16 processes, SLURM (MPMD) Details

Number of Processes: 16 ▼

Processes per Node 1 ▼

Implementation: SLURM (MPMD) Change...

srun arguments -c 16 ▼

OpenMP: 8 threads Details

Number of OpenMP threads: 8 ▼

CUDA Details

Memory Debugging Details...

Submit to Queue Configure... Parameters...

Environment Variables: none Details

Plugins: none Details

Help Options Run Cancel

DDT window



For navigation

Processing entity to control

Sparklines to quickly show variation over MPI tasks

To check the value of a variable, right-click on a variable or check the pane on the right

Parallel stack frame view is helpful in quickly finding out where each process is executing

The screenshot shows the Allinea DDT interface for Allinea Forge 7.0. At the top, a toolbar contains navigation icons (run, pause, step over, step into, step out, etc.) circled in red. Below the toolbar, the 'Current Group' is set to 'All', and a row of buttons numbered 0-15 represents MPI tasks. The main window displays the source code for 'jacobi_mpiomp.f90', with line 210 highlighted. On the right, the 'Locals' pane shows variables: 'joff' with value -43360, 'myid' with value 0, 'n' with value 23999, and 'np' with value 16. The '0' value for 'myid' is circled in red. At the bottom, the 'Stacks (All)' pane shows a parallel stack frame view with columns for Processes, Threads, and Function. The function 'set_bc (jacobi_mpiomp.f90:210)' is highlighted for process 16. The status bar at the bottom indicates 'Ready Connected to: (via tunnel) mom2:4201 -> nid10363'.

Navigation



- **Play/Continue**
- **Pause**
- **Add Breakpoint**
- **Step Into**
 - To next line; if it's a function call, enter the function
- **Step Over**
 - To next line in the current stack frame even if it's a function call
- **Step Out**
 - Return to the caller function
- **Run To Line**

Breakpoints, watchpoints and tracepoints



- **Breakpoint**
 - Stops execution when a selected line (breakpoint) is reached
 - Double click on a line to create one; there are other ways, too
- **Watchpoints for variables or expressions**
 - Stops when a variable or an expression changes its value
- **Tracepoints**
 - When reached, prints what lines of codes is being executed and the listed variables
- **Can add a condition for an action point**
 - Useful inside a loop
- **Can be active or inactive**

Many ways to check variables



- Right click on a variable for a quick summary
- Variable pane
- Evaluate pane
- Display variable values over processes (Compare across processes) or threads (Compare across threads)
- MDA (Multi-dimensional Array) Viewer
 - Visualization
 - Statistics

The screenshot displays the Multi-Dimensional Array Viewer interface. The top section shows the Array Expression set to 'uNorth[S]' and the Distributed Array Dimensions set to 'None'. The Range of \$i is set from 0 to 49, and the Display is set to 'Rows'. The central visualization shows a 3D surface plot of the data. The bottom right pane displays the following statistics:

Count:	50
Not shown:	0
Errors:	0
Aggregate:	0
Numerical:	50
Sum:	15.2618
Minimum:	0
Maximum:	0.580318
Range:	0.580318
Mean:	0.305235
Variance:	0.0304291
nan:	0
-nan:	0
inf:	0
-inf:	0
<0:	0
=0:	1
>0:	49

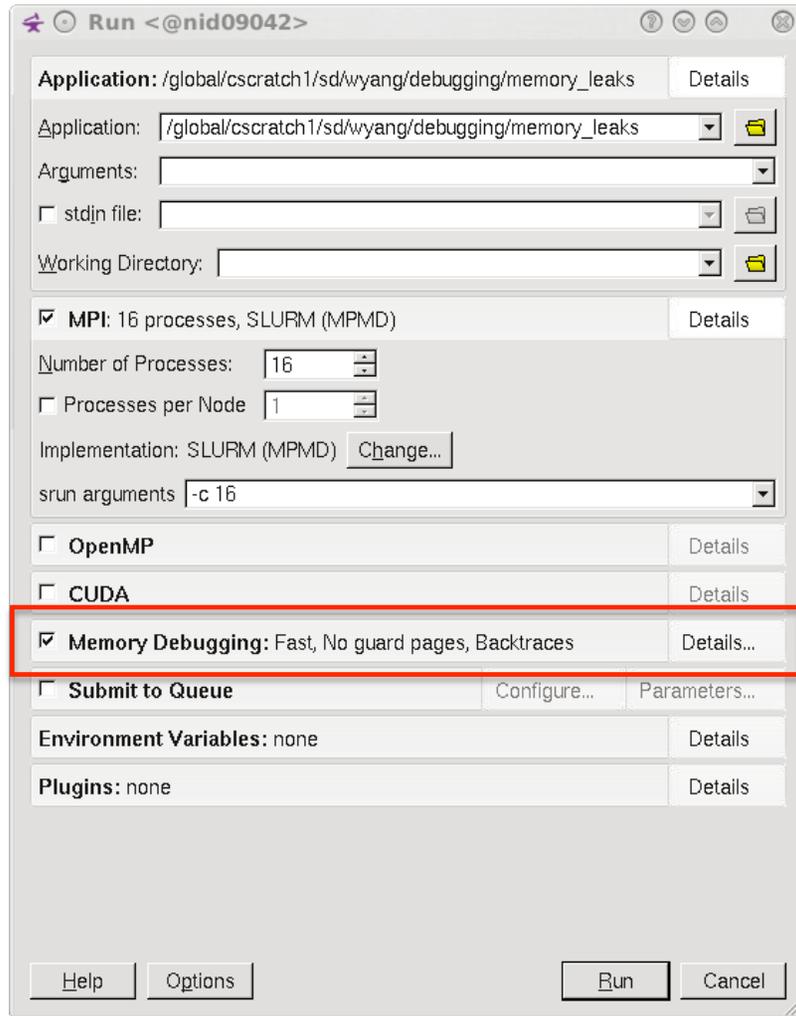
Memory debugging



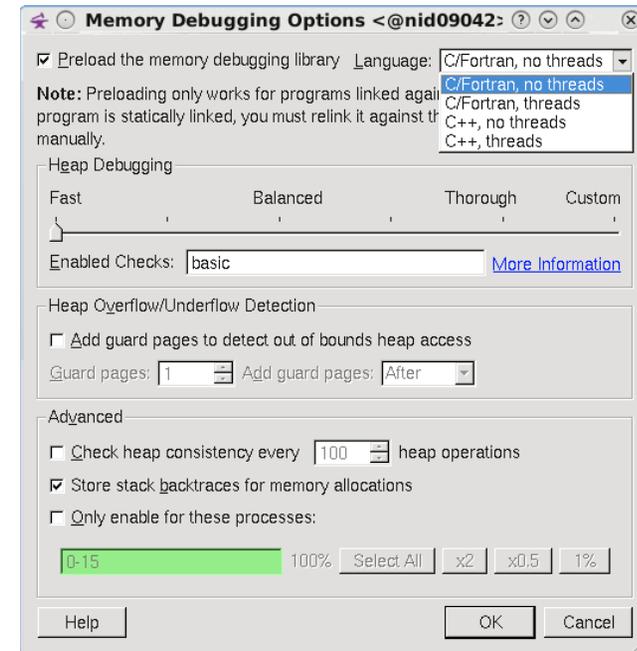
- **Why?**
 - To detect memory leaks
 - To catch out-of-bound array references
 - To catch other memory errors (“double free”, etc.)
 - To see memory usage
- **For a statically-linked executable**
 - For non-threaded code

```
$ ftn -c -g -O0 myprog.f
$ static_linking_ddt_md ftn -o myprog myprog.o
#      instead of      ftn -o myprog myprog.o
```
 - **static_linking_ddt_md_th** for threaded program
 - Similarly for C and C++ codes
 - `static_linking_ddt_md` and `static_linking_ddt_md_th` are utility scripts provided by NERSC
- **For a dynamically-linked executable, build as usual**

Enabling memory debugging



When you click 'Details...'



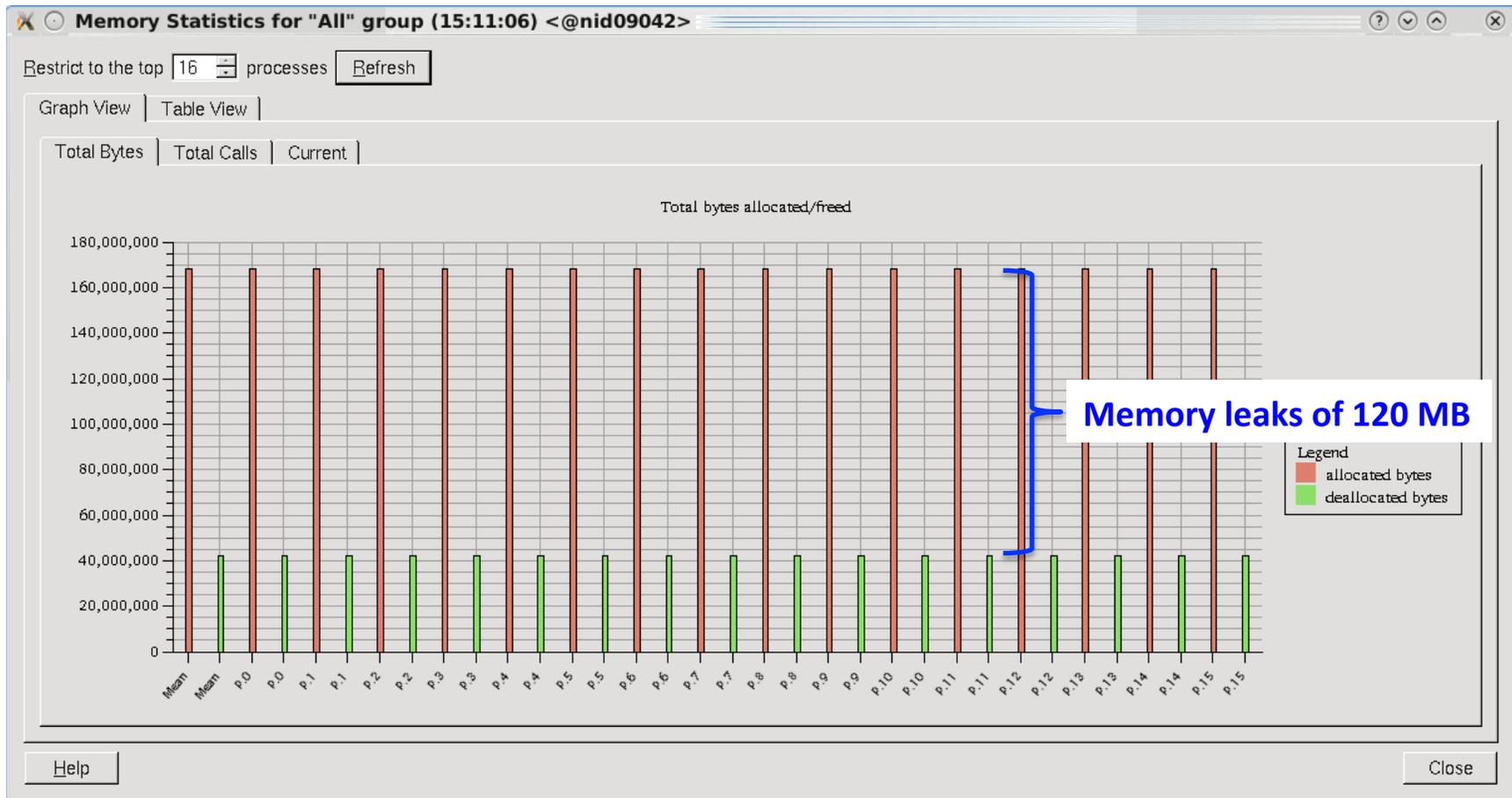
- **For a dynamically-linked binary only**
 - Check 'Preload the memory debugging library'
 - Select the appropriate one from the 'Language' pull-down menu
- **Adding guard pages (default: 4 KB) before or after memory blocks for detecting out-of-bound heap array references**

Memory debugging – Overall Memory Stats



Tools > Overall Memory Stats

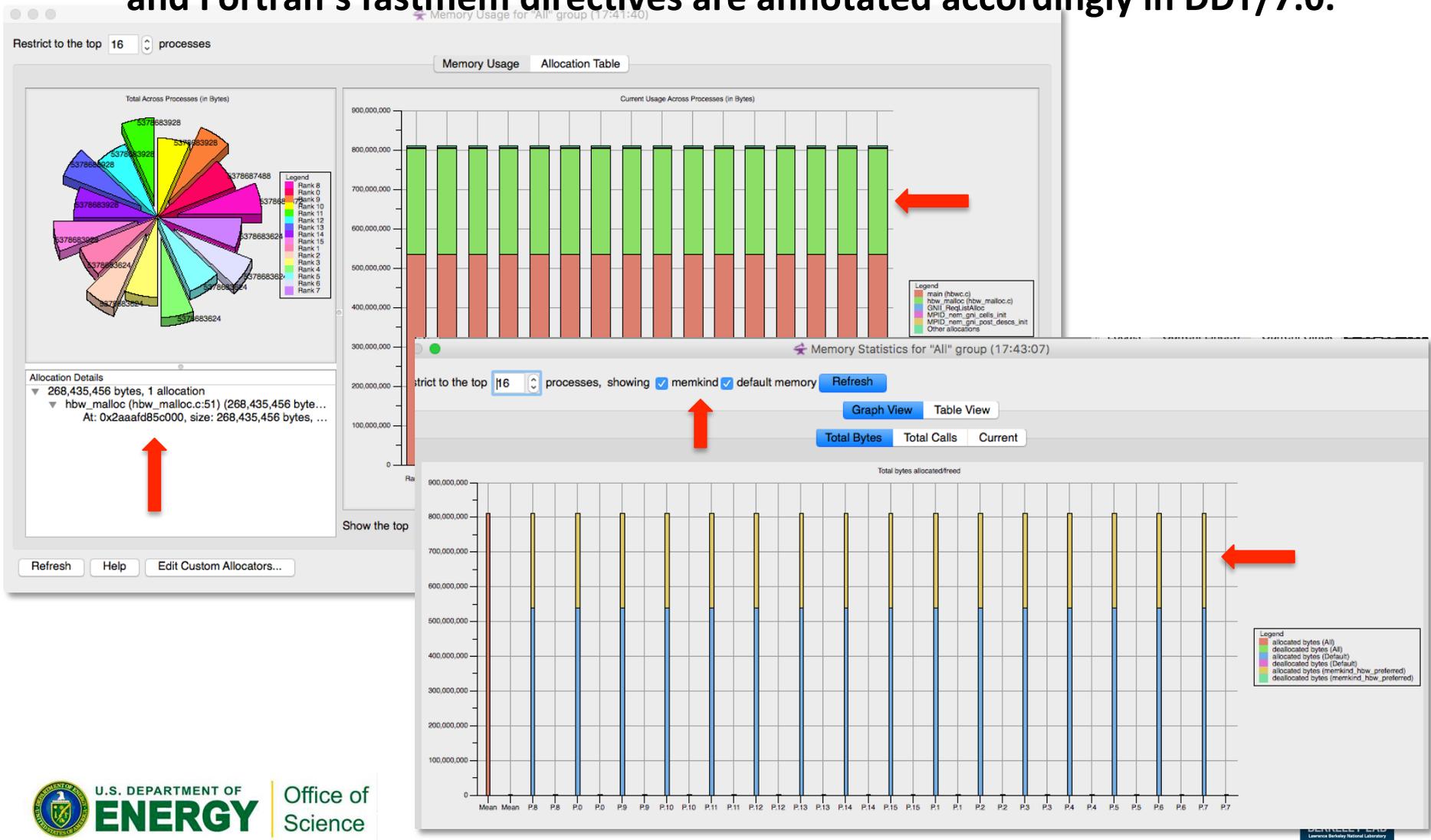
memory_leaks.f from NERSC DDT web page



KNL MCDRAM usage on Cori



- Memory blocks allocated in MCDRAM with memkind's hbw_malloc calls and Fortran's fastmem directives are annotated accordingly in DDT/7.0.



KNL MCDRAM usage on Cori (Cont'd)



- **With numactl**

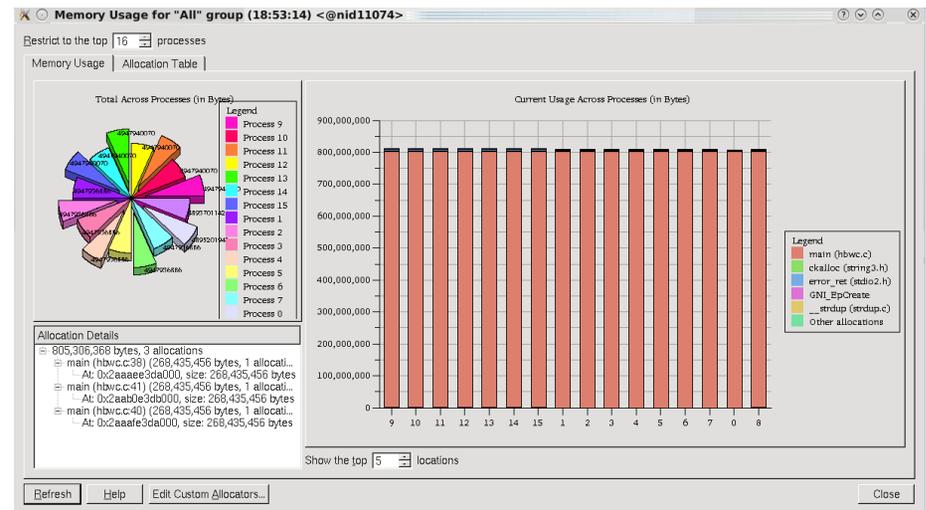
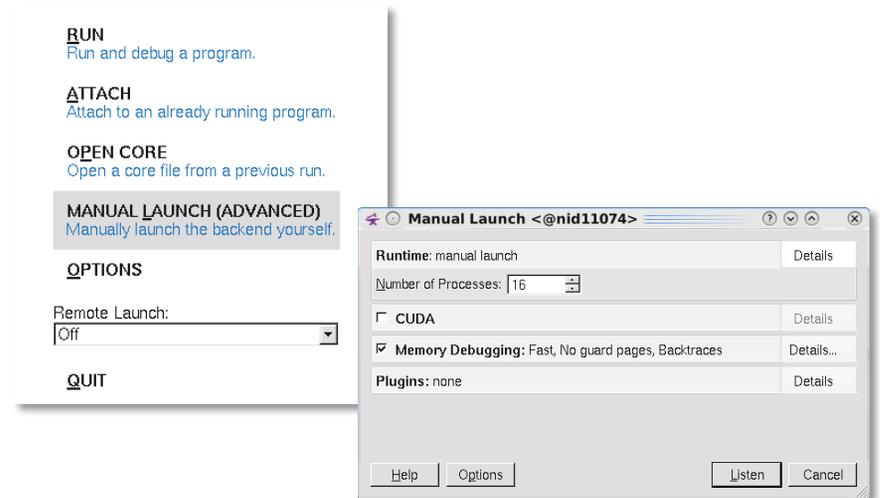
- In an interactive batch job:

1. Run ddt in background
2. Select 'MANUAL LAUNCH (ADVANCED)'
3. Set run parameters and check 'Memory Debugging'
4. Click 'Listen'
5. Run a srun command:

```
$ srun -n ... numactl \  
--preferred=1 \  
allinea-client ./a.out
```

- `--mem_bind=...`: simply use srun's `--mem_bind=map_mem:...` instead

- MCDRAM usage is not properly annotated in version 7.0. Reported to Allinea. This problem will be resolved with a new version of Slurm.



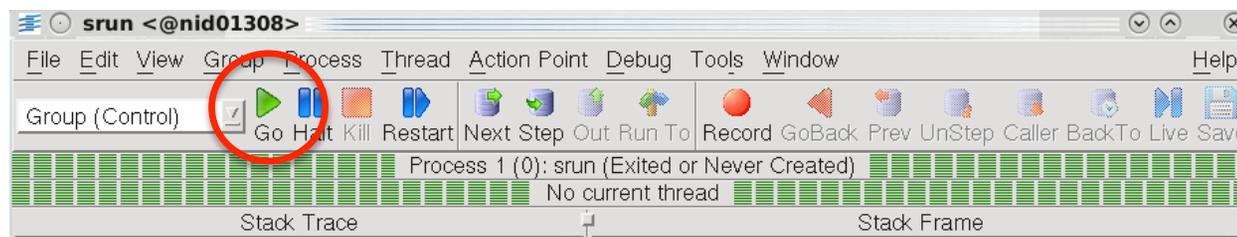
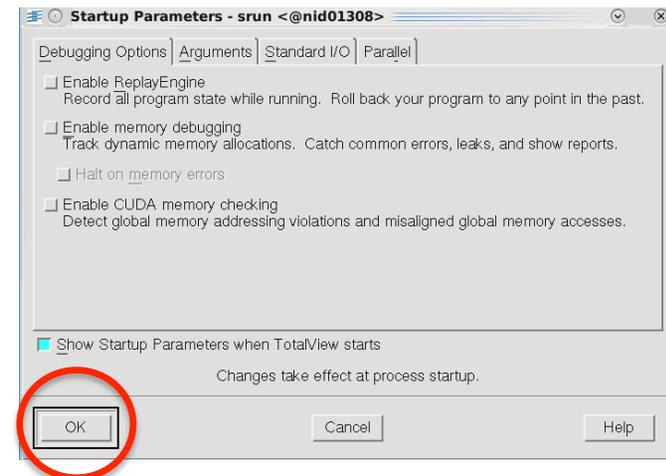
TotalView



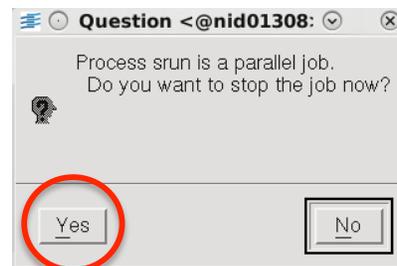
```
$ salloc -N 1 -t 30:00 -p debug
$ module load totalview
$ export OMP_NUM_THREADS=6
$ totalview srun -a -n 4 ./jacobi_mpiomp
```

Then,

- Click OK in the 'Startup Parameters - srun' window
- Click 'Go' button in the main window



- Click 'Yes' to the question 'Process srun is a parallel job. Do you want to stop the job now?'



TotalView (cont'd)



Root window

Process State	Procs	Threads	Members
Running	1	1	p1
<unknown address>	1	4	p1.1-4
-1.1	1	1	p1.1
-1.2	1	1	p1.2
-1.3	1	1	p1.3
-1.4	1	1	p1.4
Breakpoint	4	4	0-3
jacobi_mpiomp	4	4	0-3.1
-2.1	1	1	0.1
-3.1	1	1	1.1
-4.1	1	1	2.1
-5.1	1	1	3.1
sched_yield	4	20	0-3.2-6
-2.2	1	1	0.2
-2.3	1	1	0.3
-2.4	1	1	0.4
-2.5	1	1	0.5
-2.6	1	1	0.6
-3.2	1	1	1.2
-3.3	1	1	1.3
-3.4	1	1	1.4
-3.5	1	1	1.5
-3.6	1	1	1.6
-4.2	1	1	2.2

State of MPI tasks and threads; members denoted roughly as 'rank.thread'

Process window

For navigation

Stack Trace

Address	Function	FP
f90	jacobi_mpiomp,	FP=7fffffff65f0
	main,	FP=7fffffff6690
c	__libc_start_main,	FP=7fffffff6750
	_start,	FP=7fffffff6760

Stack Frame

Function "jacobi_mpiomp":
No arguments.
Local variables:

```

ierr:      0 (0x00000000)
nbr_up:    1 (0x00000001)
nbr_down: -1 (0xffffffff)
jel:      5999 (0x0000176f)
js1:      1 (0x00000001)
je:       5999 (0x0000176f)
is:       0 (0x00000000)
    
```

Function jacobi_mpiomp in jacobi_mpiomp.f90

```

65      unew(i,j) = omega * utmp + (1. - omega) * u(i,j)
66      enddo
67      enddo
68      !$omp end parallel do
69
70      call set_bc(unew,n,js,je)
71
72      ! Compute the difference between unew and u.
73
74      call compute_diff(u,unew,n,js,je,diffnorm)
75
76      if (myid == 0) print *,
77
78      ! Make the new value the c
79
80      !$omp parallel do
81      do j=js-1,je+1
82      u(:,j) = unew(:,j)
83      end do
84      !$omp end parallel do
    
```

Breakpoints, etc.

To see the value of a variable, right-click on a variable to "dive" on it or just hover mouse over it

For selecting MPI task and thread

Viewing variables

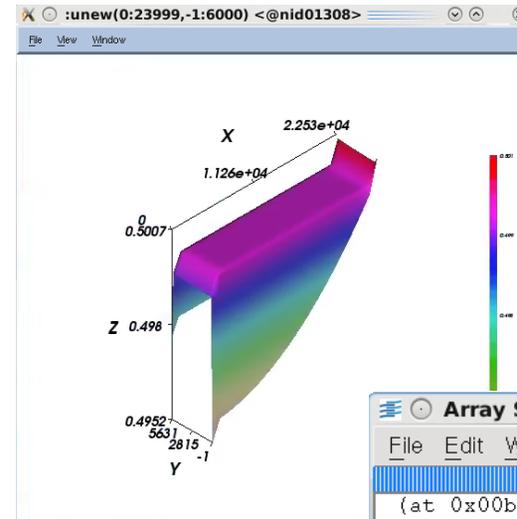


- Variable window

Variable window for 'unew' showing expression, address, slice, filter, and a table of values.

Field	Value
(0,-1)	0
(1,-1)	0
(2,-1)	0
(3,-1)	0
(4,-1)	0
(5,-1)	0
(6,-1)	0
(7,-1)	0
(8,-1)	0

- Visualization and stats



Tools > Visualize

Tools > Statistics

Array Statistics window for 'unew' showing statistical data.

Count:	144048000
Zero Count:	48001
Sum:	71995547.0228253
Minimum:	0
Maximum:	1.06248438358307
Median:	0.5
Mean:	0.49980247572215
Standard Deviation:	0.01118084478138
First Quartile:	0.5
Third Quartile:	0.5
Lower Adjacent:	0.5
Upper Adjacent:	0.5
NaN Count:	0
Infinity Count:	0
Denormalized Count:	0
Checksum:	31490

Memory debugging with MemoryScape



- **MemoryScape integrated into TotalView for memory debugging**
 - Memory leaks
 - Memory usage
 - Memory corruption
 - ...

- **A statically-linked executable**

```
$ module load totalview  
$ CC -g -O0 -o memry_leaks memory_leaks.o `${TVMEMDEBUG_POST_OPTS}
```

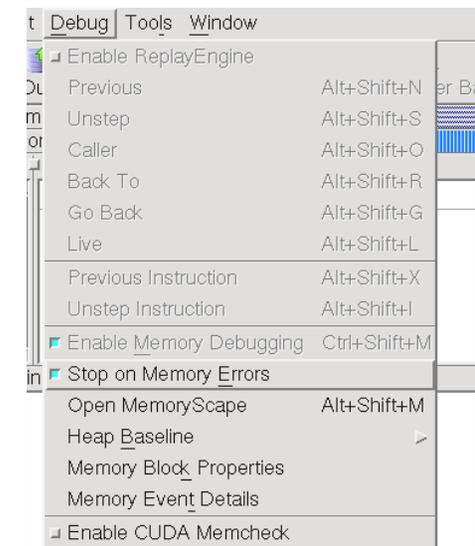
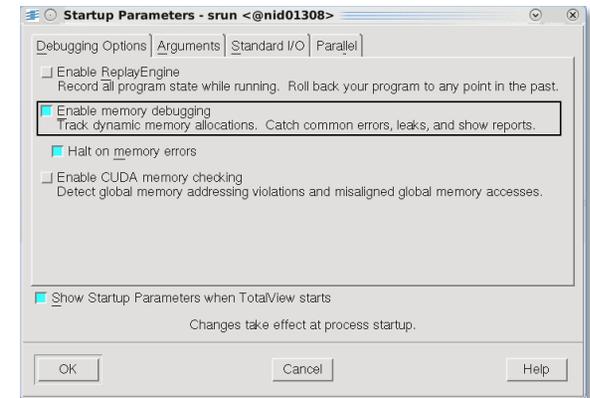
- **A dynamically-linked executable, build as usual**

```
$ CC -dynamic -g -O0 -o memry_leaks memory_leaks.o
```

Memory debugging with MemoryScape



- Start TotalView and enable memory debugging in the 'Startup Parameters' window
- Proceed to use TotalView as usual
- For memory-related issues, open MemoryScape from the Debug pull-down menu



Memory debugging examples



MemoryScope 2016.07.22 <@nid01308>

File Tools Window Help

Home | Memory Reports | Manage Processes | Memory Debugging Options | Tips

Summary | Leak Detection | Heap Status | Memory Usage | Corrupted Memory | Memory Comparisons

February 21, 2017

Save Data
Save Report...
Export Memory Data...

Leak Detection Reports
Backtrace Report

Other Reports Categories
Heap Status Reports
Memory Usage Reports
Corrupted Memory Report
Compare Memory Usage

Other
Manage Filters

Process Selection

Process: Parallel Job srunc<memory-leaks> 1

Leak Detection Source Report

Options
 Relative to Baseline Enable Filtering

Process	Bytes	Count	Begin Address	End Address	Backtrace ID	Allocat
Process 3: srunc<memory-leaks>.1	967.34KB	4767				
memory-leaks	967.34KB	4767				
myClassB.cxx	816.00KB	1560				
myClassB::init	816.00KB	1560				
Line 51	768.00KB	1536				
Block 597.1536	512	1	0x00b803cd	0x00b805bf	597	
Block 597.1535	512	1	0x00b7ffa0	0x00b8019f	597	
Block 597.1534	512	1	0x00b7fb80	0x00b7fd7f	597	
Block 597.1533	512	1	0x00b7f760	0x00b7f95f	597	

Back-trace

ID	Function	Line #	Source Information
597	malloc		memory-leaks
	myClassB::init	51	myClassB.cxx
	myClassB::myClassB	27	myClassB.cxx
	main	328	main.cxx
	_libc_start_main	242	libc-start.c
	_start		memory-leaks

Source

```

C:\scratch\1\sd\wyang\debugging\examples\src\myClassB.cxx
43 }
44 }
45 }
46 void myClassB::init(void) {
47
48     b_pp = (int**) malloc( size * sizeof(int *));
49
50     for(int i=0; i<size; i++) {
51         b_pp[i] = (int*) malloc( 128 * sizeof(int));
52     }

```

Memory Debugging Options | Tips

Memory Usage | Corrupted Memory | Memory Comparisons

February 21, 2017

Save Data
Save Report...
Export Memory Data...

Other Reports Categories
Heap Status Reports
Memory Usage Reports
Leak Detection Reports
Compare Memory Usage

Other Tasks
Manage Filters

Process Selection

Process: Parallel Job srunc<memory-corrupt> 1

Parallel Job srunc<memory-corrupt> 1

MPI_COMM_WORLD

srunc<memory-corrupt>.1

srunc<memory-corrupt>.2

Corrupted Memory Report

Options
 Enable Filtering

	Preceding Block	Corrupted Block	Following Block
1	0x00a6da30 20480 bytes 0x00a6da3f	0x00a6da30 64 bytes 0x00a6da3f	0x00a6da30 64 bytes 0x00a6da3f
2	0x00a6da50 64 bytes 0x00a6da5f	0x00a6da30 64 bytes 0x00a6da3f	0x00a6da10 64 bytes 0x00a6da1f
3	0x00a6db10 64 bytes 0x00a6db1f	0x00a6db70 64 bytes 0x00a6db7f	0x00a6dbd0 64 bytes 0x00a6dbdf
4	0x00a6db70 64 bytes 0x00a6db7f	0x00a6dbd0 64 bytes 0x00a6dbd0	0x00a6dd30 64 bytes 0x00a6dd3f
5	0x00a6dd30 64 bytes 0x00a6dd3f	0x00a6dd30 64 bytes 0x00a6dd3f	0x00a6dd10 64 bytes 0x00a6dd1f

Backtrace/Source | Memory Content

Hexadecimal Count: 88

0x00a6da60	0x0c	0x00	0x00	0x00	0x0b	0x00	0x00	0x00
0x00a6da68	0x0a	0x00	0x00	0x00	0x09	0x00	0x00	0x00
0x00a6da70	0x08	0x00	0x00	0x00	0x07	0x00	0x00	0x00
0x00a6da78	0x06	0x00	0x00	0x00	0x05	0x00	0x00	0x00
0x00a6da80	0x04	0x00	0x00	0x00	0x03	0x00	0x00	0x00
0x00a6da88	0x02	0x00	0x00	0x00	0x01	0x00	0x00	0x00
0x00a6da90	0x00	0x00	0x00	0x00	0xff	0xff	0xff	0xff

Bytes: 1 2 4 8

Corrupted guard blocks

STAT (Stack Trace Analysis Tool)



- **Gathers stack backtraces (showing the function calling sequences leading up to the ones in the current stack frames) from all (MPI) processes and merges them into a single file (*.dot)**
 - Results displayed graphically as a call tree showing the location in the code that each process is executing and how it got there
 - [Can be useful for debugging a hung application](#)
 - With the info learned from STAT, can investigate further with DDT or TotalView
- **Works for MPI, CAF and UPC, but not OpenMP**
- **STAT commands (after loading the 'stat' module)**
 - stat-cl: invokes STAT to gather stack backtraces
 - stat-view: a GUI to view the results
 - stat-gui: a GUI to run STAT or view results
- **For more info:**
 - 'intro_stat', 'stat-cl', 'stat-view' and 'stat-gui' man pages
 - https://computing.llnl.gov/code/STAT/stat_userguide.pdf
 - <http://www.nersc.gov/users/software/debugging-and-profiling/stat-2/>

Hung application with STAT



- If your code hangs in a consistent manner, you can use STAT to see if and where some MPI ranks are stuck.
- Currently, one known way to use STAT is as follows.

```
$ ftn -g -o jacobi_mpi jacobi_mpi.f90          with usual optimization flags, if any
$ salloc -N 1 -t 30:00 -p debug -C knl,quad,cache
```

```
...
```

```
$ srun -n 4 ./jacobi_mpi &
```

```
[1] 93834
```

```
$ module load stat
```

```
$ stat-cl -i 93834
```

-i to get source line numbers

STAT samples stack backtraces a few times

```
...
```

```
Attaching to application...
```

```
Attached!
```

```
Application already paused... ignoring request to pause
```

```
Sampling traces...
```

```
Traces sampled!
```

```
...
```

```
Resuming the application...
```

```
Resumed!
```

```
Merging traces...
```

```
Traces merged!
```

```
Detaching from application...
```

```
Detached!
```

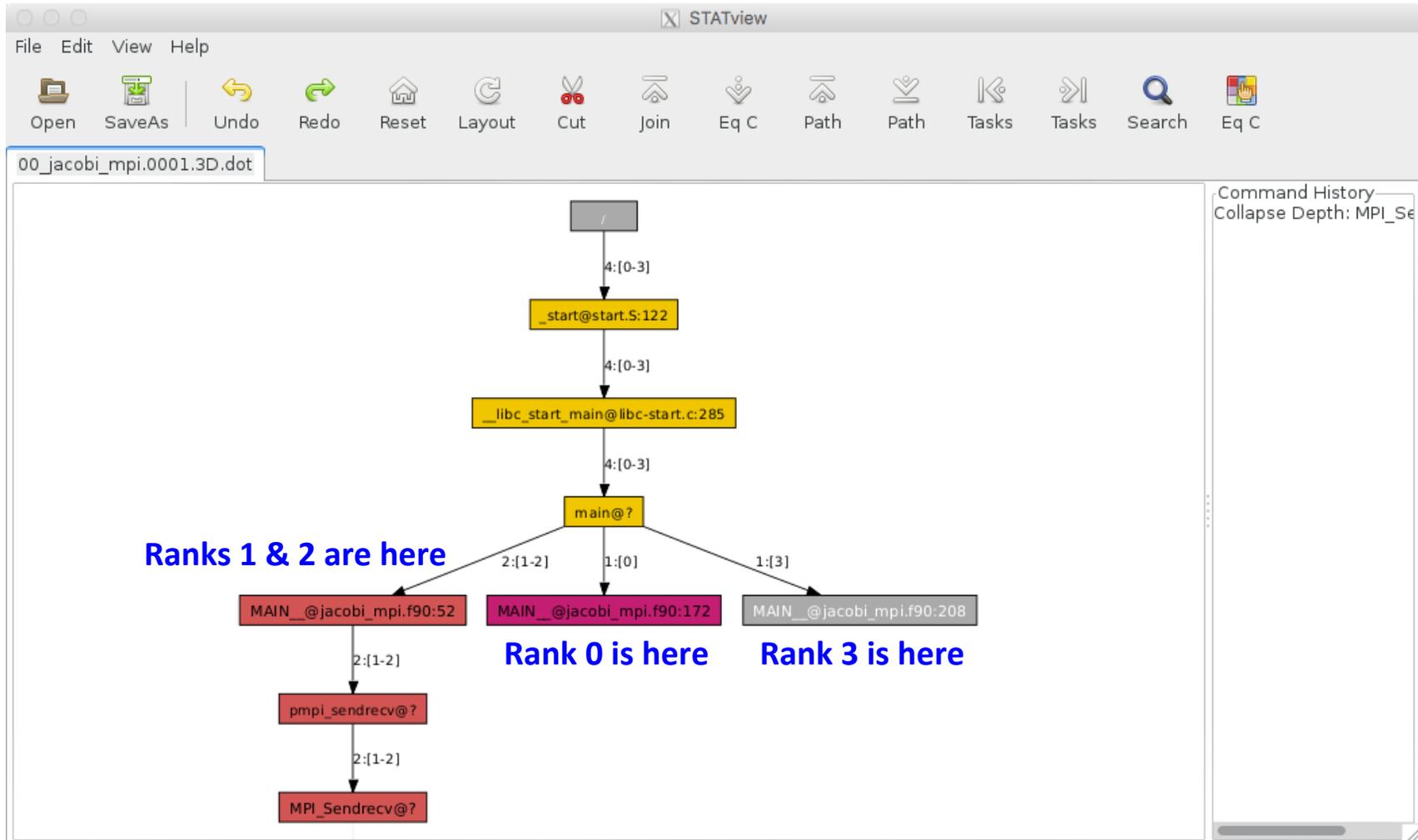
```
Results written to /global/cscratch1/sd/wyang/debugging/stat_results/jacobi_mpi.0001
```

```
$ ls -l stat_results/jacobi_mpi.0001/*.dot
```

```
-rw-r----- 1 wyang wyang 2768 Feb 20 21:24 stat_results/jacobi_mpi.0001/00_jacobi_mpi.0001.3D.dot
```

```
$ stat-view stat_results/jacobi_mpi.0001/00_jacobi_mpi.0001.3D.dot
```

Hung application with STAT (Cont'd)



ATP (Abnormal Termination Processing)



- **ATP gathers stack backtraces from all processes if an application fails**
 - Invokes STAT underneath
 - Output in atpMergedBT.dot and atpMergedBT_line.dot (which shows source code line numbers), which are to be viewed with stat-view

- **By default, the atp module is loaded on Cori and Edison, but ATP is not enabled; to enable:**

```
export ATP_ENABLED=1      # sh/bash/ksh
setenv ATP_ENABLED 1     # csh/tcsh
```

- **Can get core dumps (*core.atp.jobid.rank*), too, by setting coredumpsize unlimited:**

```
ulimit -c unlimited      # sh/bash/ksh
unlimit coredumpsize     # csh/tcsh
```

but they do not represent the exact same moment in time (therefore the location of a failure can be inaccurate)

- **For more info**
 - ‘intro_atp’ man page
 - <http://www.nersc.gov/users/software/debugging-and-profiling/stat-and-atp/>

Hung application with ATP



- Force to generate backtraces from a hung application
- For the following to work, must have used
 - ‘export ATP_ENABLED=1’ in batch script
 - ‘export FOR_IGNORE_EXCEPTIONS=true’ in batch script for Intel Fortran
 - ‘-f no-backtrace’ at compile/link time for GNU Fortran

```
$ sacct -j 4097861 Find the job step ID
```

```
JobID JobName Partition Account AllocCPUS State ExitCode
```

```
-----
...
4097861.0 jacobi_mp+ nstaff 4
```

```
$ ssh edimom02 Kill the application on a MOM node
```

```
$ scancel -s ABRT 4097861.0
```

```
$ exit
```

```
$ cat slurm-4097861.out
```

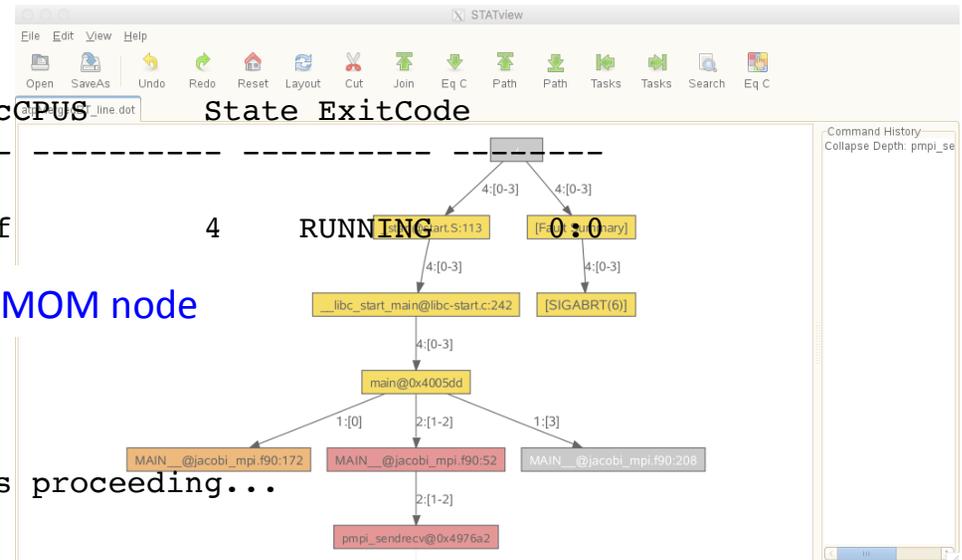
```
Application 4097861 is crashing. ATP analysis proceeding...
```

```
...
Process died with signal 6: 'Aborted'
```

```
View application merged backtrace tree with: stat-view atpMergedBT.dot
```

```
$ module load stat
```

```
$ stat-view atpMergedBT.dot # or statview atpMergedBT_line.dot
```



- Suite of debugging and profiler tools
- Tools include
 - **memcheck**: memory error and memory leaks detection
 - **massif, dhat (exp-dhat)**: heap profilers
 - **cachegrind**: a cache and branch-prediction profiler
 - **callgrind**: a call-graph generating cache and branch prediction profiler
 - **helgrind, drd**: pthreads error detectors
- For info:
 - <http://valgrind.org/docs/manual/manual.html>

Valgrind's memcheck



```
$ module load valgrind
$ ftn -dynamic -g -O0 memory_leaks.f $VALGRIND_MPI_LINK
$ salloc -N 1 -t 30:00 -p debug -C knl
$ srun -n 2 valgrind --leak-check=full --log-file=%p ./a.out
$ ls -l
...
-rw-r--r-- 1 wyang wyang      7550 Feb 21 23:36 91835
-rw-r--r-- 1 wyang wyang      7550 Feb 21 23:36 91836
```

Could have explicitly added '--tool=memcheck'

- Let's look at the report for process 91835

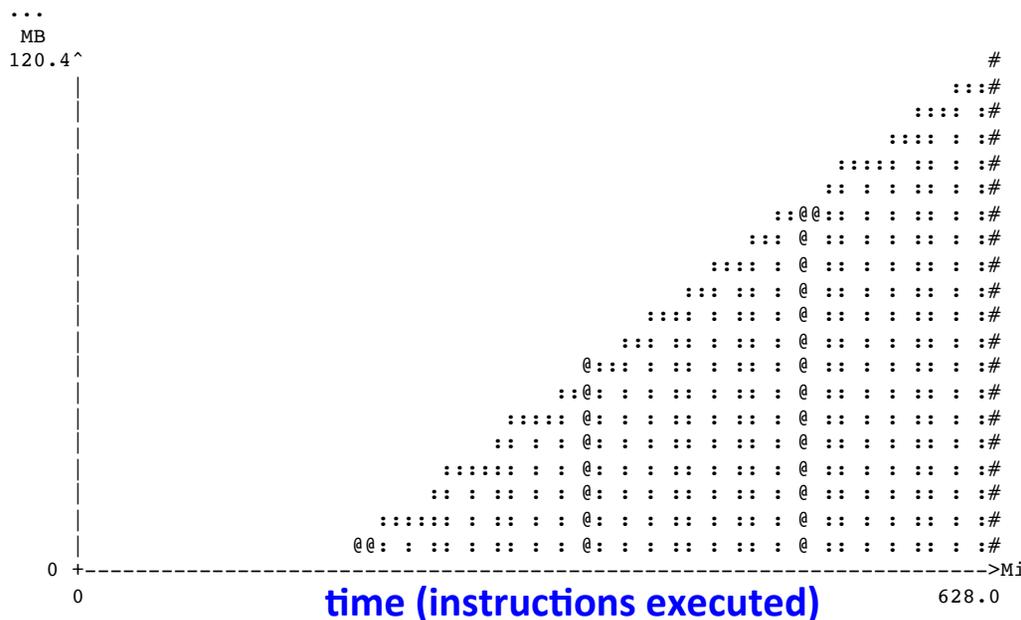
```
$ more 91835
...
==91835== LEAK SUMMARY:
==91835==    definitely lost: 83,886,880 bytes in 20 blocks
==91835==    indirectly lost: 0 bytes in 0 blocks
==91835==    possibly lost: 41,943,440 bytes in 10 blocks
==91835==    still reachable: 103,903 bytes in 74 blocks
==91835==    suppressed: 0 bytes in 0 blocks
...
```

- Can suppress spurious error messages by using a suppression file (--suppressions=/path/to/directory/file)

Valgrind's massif

- For profiling heap memory usage

```
$ ftn -g -O2 memory_leaks.f
$ srun -n 2 -c 128 valgrind --tool=massif ./a.out
$ ls -lrt
...
-rw----- 1 wyang wyang 50233 Feb 21 23:55 massif.out.92841
-rw----- 1 wyang wyang 81113 Feb 21 23:55 massif.out.92842
$ ms_print massif.out.92841
```



'#': normal snapshot; basic info provided

'@': detailed snapshot where detailed info is provided

'#': peak snapshot where the peak heap usage is

This example strongly suggests memory leaks

```
Number of snapshots: 95
Detailed snapshots: [14, 29, 44, 48, 50, 51, 61, 71, 81, 91 (peak)]
...
```




National Energy Research Scientific Computing Center